Tuning Object-centric Data Management Systems for Large Scale Scientific Applications

Houjun Tang, Suren Byna, Stephen Bailey, Zarija Lukić,
Jialin Liu, Quincey Koziol, Bin Dong
Lawrence Berkeley National Laboratory
Berkeley, California 94720
{htang4, sbyna, stephenbailey, zarija, jalnliu, koziol, dbin}@lbl.gov

***Abstract*—Efficient management of scientific data on high-performance computing (HPC) systems has been a challenge, as it often requires knowledge of various hardware and software components of the system, as well as tedious manual effort in optimizing parallel I/O for each application. This situation is exacerbated by the fact that storage systems on upcoming exascale supercomputers are equipped with an unprecedented level of complexity due to a deep storage and memory hierarchy with heterogeneous hardware and their management software. Simple and effective data management methods are critical for numerous scientific applications that are storing and analyzing massive amounts of data on HPC systems. Object-centric data management systems (ODMS) provide an easy-to-use interface, allow for massive scalability with relaxed consistency, and have been gaining popularity in the HPC community. However, tuning an ODMS to achieve its full potential on existing HPC systems with large-scale science use cases still remains a challenging task. In this paper, we explore and evaluate various well-known I/O tuning techniques on a new ODMS called Proactive Data Containers (PDC). Our experiments using real science applications and I/O kernels demonstrate that the benefits of these tuning methods with up to $9X$ I/O performance speedup over the previous version of PDC, and $47X$ over a highly optimized HDF5 implementation.**

I. Introduction

Large scale scientific simulations, experiments, and observations that use HPC systems are producing and/or analyzing data that amounts to tens of terabytes or even several petabytes, and this volume of data is projected to increase even further [1]. To reduce the I/O latency in accessing data from disk-based storage systems, HPC systems are being deployed with SSD-based storage either on each compute node or shared among all the compute nodes, or both. Taking advantage of these new storage architectures that contain heterogeneous devices managed by different software layers is a challenging and time-consuming task for users.

Numerous I/O performance tuning techniques have been proposed for existing storage systems such as Lustre [2] and GPFS [3] that are file-based and follow the POSIX-IO standards. However, these techniques have to be POSIX-IO compliant, resulting in limited performance improvements as the number of processes performing I/O on the same file system scale up to the millions. This is due to POSIX-IO's requirement of strong consistency, which is often not necessary in HPC environments, as scientific applications rarely write to the same part of a file by multiple processes concurrently.

Object-based storage systems such as Amazon S3 [4], and OpenStack Swift [5] have been deployed and used by cloud service providers, and are able to serve a large number of clients efficiently. Several recent efforts have been proposed to design and develop object-based storage systems for HPC environment, such as Ceph RADOS [6] and DAOS [7], that tackle the challenges posed by large data volume and heterogeneous architectures.

We have recently developed the Proactive Data Containers (PDC) [8], [9], a user-level object-centric data management system (ODMS). PDC provides an object-centric interface to perform efficient data management operations at the object granularity utilizing existing HPC system's software and hardware components, which abstracts away the complexity to manage files across multiple layers of the memory hierarchy. We have shown that object-centric data management with PDC achieves better performance compared to existing POSIX-compliant I/O libraries using several benchmarks [8].

However, in using PDC for large-scale science use cases of managing data produced by cosmology simulations and observations, we have uncovered various challenges and performance improvement opportunities that are common to ODMS runtime systems. The three challenges we identified in an ODMS managing data at large scale are: (1) with an ODMS that has I/O servers co-exist with the application, and actively manage and move data across the memory hierarchy, I/O performance can vary significantly with different requests and ODMS server/client configurations. (2) while data caching and prefetching are well-known optimization technologies in reducing data access latency of memory or storage, strategies that can take advantage of the additional knowledge from an ODMS have not yet been explored. (3) While ODMS technologies can tolerate relaxed POSIX-IO consistency, new methods that coordinate the data and metadata operations to reduce the I/O latency have not been explored.

We present in this study our solutions to these challenges, where we enhanced the PDC system with a runtime decision making method to assign workloads, using object relationships and data access patterns to guide prefetching and caching across the memory hierarchy, and enabling asynchronous

2640-0316/19/$31.00 ©2019 IEEE
DOI 10.1109/HiPC.2019.00023

communication among servers and clients to achieve high performance with relaxed consistency. We chose to implement these new tuning techniques in the PDC system, as its data and the metadata management services are deployed at the user-level, and can be easily added and enabled. Additionally, PDC allows users to utilize various underlying storage devices, such as SSD-based burst buffer and disk-based file systems, as PDC's back-end data storage in their preferred HPC systems. In contrast, other HPC-oriented object storage systems (such as DAOS, Ceph RADOS, etc.) require a system-wide change to the storage architecture implementation. In summary, contributions of this study are:

- A dynamic I/O aggregation method for an ODMS that automatically assigns the I/O workload to aggregators and/or to application clients, based on the number of available clients, I/O aggregators, access pattern, and data distribution in storage devices.
- A multi-level prefetching and caching strategy to utilize all resources across the deep memory hierarchy based on the object relationships and I/O access pattern.
- A relaxed consistency method through asynchronous operations with aggregation both on clients and on ODMS servers.

We have evaluated our optimization strategies using a large-scale cosmology simulation, a space geometry survey dataset with millions of objects, and two I/O kernels from a particle-in-cell code and its analysis application. These evaluations are performed on a Cray XC40 system located at the National Energy Research Scientific Computing Center (NERSC). We show that using our proposed techniques, the PDC achieves a speedup of up to $9X$ compared to its previous non-optimized version and up to $47X$ faster than existing POSIX-compliant I/O library, i.e., HDF5.

In the remainder of the paper, we discuss requirements of an ODMS in the HPC environment and challenges faced by these systems, and present our proposed solutions and an implementation using the PDC system. We then evaluate the performance of our proposed optimizations and related work before concluding the paper.

II. Scope of ODMS Optimizations

In general, an ODMS can manage any type of data, from small objects, such as human-readable texts, to large multi-dimensional arrays. In this section, we describe the crucial optimizations needed in object-centric storage for large scale scientific applications on HPC systems. They are categorized into data movement optimizations, metadata management optimizations, and those common to both data and metadata management.

A. Data Movement Optimization

1) Optimized write with log-structured storage: Log-structured storage [10], [11] organizes data in an append-only manner, such that whenever there is new data that needs to be written, instead of seeking to an offset location in the file and to a corresponding location on the storage device, the data is simply appended to the end. With this approach, data is always written sequentially rather than randomly, which would significantly improve the write performance. It has been used by I/O libraries such as PLFS [12] and ADIOS [13] that demonstrated performance advantages.

The log-structured approach can also be used by an ODMS, as it offers superior performance even when writing with millions of small-sized objects, and can be implemented on top of existing file systems. It also makes the data immutable, guarantees that no data is overwritten, and the current version of an object is defined to be the most recently written. As new versions of objects are written to the log, the previous versions of those objects are rendered obsolete. Thus, it requires periodic garbage collection to reclaim storage space. However, data in log-structured format brings challenges to efficient data read operations, especially when the data read requests differ from the writes, resulting in a large number of costly non-contiguous accesses.

2) Caching and prefetching in a deep memory hierarchy: With multiple layers of memory and storage devices being deployed in upcoming supercomputing systems, such as the NVRAM-based burst buffer and node-local SSDs. Given the user's data access requests, moving data to/from the closest layer would provide the lowest latency, but requires a proactive management of the ODMS. Caching and prefetching are effective technologies that are widely used, and with the additional information managed by the ODMS, more accurate decisions can be made in a timely manner. The layered storage devices offer an opportunity for overlapping I/O with computation. For read operations, an ODMS can prefetch and cache data in different layers automatically and transparently, without user's involvement. This improves productivity by allowing scientists to focus on science rather than on managing data. For write operations, applications can write data to a temporary location on faster storage devices and have the ODMS to move the data automatically to the longer-term storage system.

3) Fine-grained data access: As scientific data is often stored as multi-dimensional arrays, accessing a sub-region of an array is a common practice. This is a major contrast from cloud computing object storage, where an object is always accessed in its entirety. Reading a big array object for using a small portion of data is unnecessary and costly. Moreover, techniques such as adaptive mesh refinement (AMR), produce large volumes of data with a hierarchical, multi-level, and multi-resolution data structure. AMR only refines the grid in regions where a high resolution is necessary, leaving other parts empty on finer levels. Although it is possible to map each AMR block or cell to an object, however, it could lead to a drastic increase in the number of objects (millions for only one timestep data), as well as the amount of duplicated metadata associated with each block or cell. To provide efficient fine-grained data access, HPC-oriented ODMS requires splitting of an object into regions when necessary, and providing convenient subset data access APIs in addition to the entire object access methods. Each region is immutable and the entire object may include or remove any regions dynamically.

B. Metadata Management Optimization

1) Flat namespace: Traditional file systems organize files into hierarchies (directories and sub-directories). Each file is accessed with its full path, which could be lengthy and often require users to store in a manifest file or a database. On the other hand, an ODMS removes the folder hierarchy entirely, every object is stored in a flat namespace. Users should still able to group similar objects, by either creating containers that include the desired objects, or adding unique attributes to the object metadata.

2) Searchable metadata: Parallel file systems such as Lustre, GPFS, and NFS typically associate each file with predefined metadata attributes, which typically contains system information, such as user ID, timestamp, access permission, etc. Such information, however, is rarely useful to help users find the target data for knowledge discovery. In addition, users have limited or no ability to modify or add additional information to the file system metadata, leaving them to manage their user-defined metadata in an ad-hoc manner. Although I/O libraries such as HDF5 offer the ability to store data and user-defined metadata within the same file container, searching those data still requires manual work and hand optimization.

Object-centric storage systems allow attaching an extensive amount of user-defined metadata to data objects. With metadata being extensible, users can add any information they desire to an object, including data provenance or analysis results in the metadata. In addition to allowing rich and extensible metadata, object-centric storage also supports searching the metadata. Although each object has a unique ID that can be used for direct access, a more productive and efficient way to interact with objects is through metadata search. Users only need to provide a few descriptions (metadata attributes) of the data and the object-centric storage system would automatically retrieve the corresponding data.

C. Common Optimization to Data and Metadata Management

1) Eventual or strong consistency: To ensure high performance, a data object in an ODMS is often "eventually consistent" [5], which means a reader may not get the latest version of an object's data while another writer is updating, or just completed an update. Such relaxed consistency can improve the performance of scientific applications that produce data and only access the data during subsequent post-processing workflows.

In the case when the user demands a strong consistency, locks can be used that temporarily queues the read request until the write is finished. However, such locking mechanism can introduce additional overhead. It is desirable to have a system to support both consistency model and allowing the users to select one based on their needs.

2) Asynchronous operation: Scientific data analysis often includes several phases, resulting in complex workflows and data movement among different tasks. Supporting asynchronous data movement and metadata operations would greatly benefit the data management in these workflow systems, especially when data is expressed as objects. Asynchronous operations allow various tasks of a workflow to utilize the computing resources. For instance, with an asynchronous write, a workflow engine may continue with the next task without waiting for the write operation to complete. The system should also be scalable with additional nodes without performance degradation. This can be achieved with the entire system running in user-space, and making it easy to adjust the service processes based on the workload.

III. Tuning Object-centric Data Management

An ODMS hides the complexity to manage a large number of files in different memory and storage layers from the users. With the capability to manage both user-defined and system collected metadata, it can proactively move the data across the memory hierarchy with prefetching and caching, provide scalable data and metadata operations that move beyond POSIX I/O semantics with a relaxed consistency model. However, existing approaches have not yet been able to express and utilize these optimization strategies. In this section, we describe various data and metadata optimization methods that we identified as suitable tuning approaches in Section II to improve the data access efficiency of an ODMS.

A. Data Movement Optimization

As data is stored and managed as objects in an ODMS, data access patterns can be very different from those observed in existing parallel file systems. Small data object access is more frequent and on a large scale.

1) Log-structured storage and I/O aggregation: As mentioned Section II-A1, log-structured data store provides an efficient way to offload data from applications' memory to the storage system. However, reading data from log-structured format efficiently is challenging due to a potentially large number of non-contiguous access. For example, a dataset generated by a dark matter survey, called BOSS (described in detail in Section VI-A), has ≈ 2.6 million objects, each relatively small in size (around $1MB$), and is accessed through metadata queries. These queries often result in a large number of non-contiguous small data access, which is known to be slow in existing disk-based storage devices. Though faster SSD-based storage devices improve performance in these cases, they are typically used for temporary storage. Disk-based storage still has to be accessed either to cache data into the SSDs or to read the data into memory.

I/O aggregation can be helpful in optimizing the read performance with log-structured format [10]. It adds a layer between the computing subsystem and storage subsystems in the HPC environment that aggregates I/O requests from many processes to fewer dedicated I/O processes. The I/O aggregators have the opportunity to reorder and merge the requests for faster data access. The dedicated I/O processes may either be the user-space data movement processes (e.g., in [8]) or those running on dedicated I/O nodes (IBMs Blue Gene systems).

Although I/O aggregation (sometimes referred to as "I/O forwarding") has been proven to be effective in traditional file

I/O [14], [15], [16], it has not yet been applied and evaluated with an ODMS. Toward the application of I/O aggregation in ODMS, we first evaluated its efficiency for different data access patterns, by developing an I/O benchmark that performs read operations with three types of access: 1) contiguous data accesses, 2) non-contiguous data accesses from a small number of storage servers, and 3) non-contiguous data access from a large number of storage servers. For each access pattern, we measured the I/O time by varying individual I/O request sizes and the number of processes.

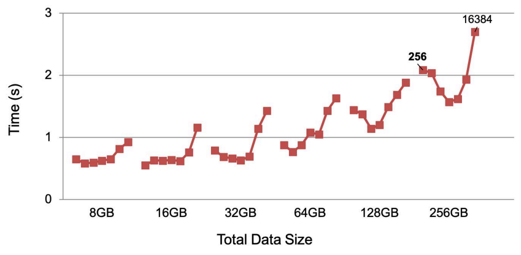
Figures 1a to 1c show the I/O time measured with the three patterns, respectively. For the contiguous access pattern (Figure 1a), each process reads one large chunk of data, and fewer number of readers result in better performance for a larger amount of data, but the performance variation with different number of processes is insignificant as overall time is short. Whether to enable I/O aggregation in this case requires other considerations such as communication cost and if fast data transfer (e.g., through shared memory) is available. On the other hand, with the two non-contiguous access patterns (Figures 1b and 1c), where each process reads a number of non-contiguous $1MB$ chunks, the performance depends on the number of storage servers involved. In the case where the requested data are from only a few storage servers and there aren't many overlapping requests among different reader processes, it is better to have more readers, up to a certain extent. However, when the readers are retrieving data from mostly same storage servers, and the number of processes is larger (i.e., more than 4096), it results in I/O congestion and increasing the number of readers leads to worse performance.

$$\begin{cases} \text{Case 1: } N_{nc} < N_{nc}^{thrs} \wedge T_{overhead} > T_{overhead}^{thrs} \\ \text{Case 2: } N_{nc} > N_{nc}^{thrs} \wedge N_{tgt} < N_{tgt}^{thrs} \\ \text{Case 3: } N_{nc} > N_{nc}^{thrs} \wedge N_{tgt} > N_{tgt}^{thrs} \wedge \\ \qquad (N_{fwd} \ll N_{con} \vee N_{clt} > N_{con}) \end{cases}$$

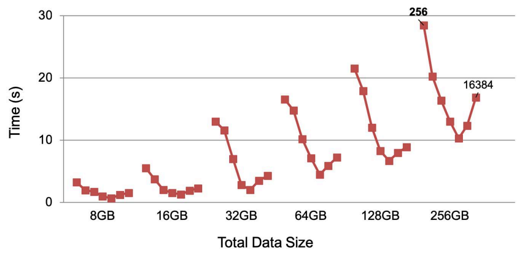
Symbol	Description
N_{nc}	Number of non-contiguous accesses.
N_{nc}^{thrs}	Threshold for non-contiguous access category (default: 10).
$T_{overhead}$	Overhead to transfer data between I/O aggregators and clients.
$T_{overhead}^{thrs}$	Overhead threshold for inefficient data transfer.
N_{tgt}	Number of storage servers the data objects are from.
N_{tgt}^{thrs}	Threshold for large number of storage servers (default: 10).
N_{fwd}	Number of I/O aggregator processes.
N_{clt}	Number of application (client) processes.
N_{con}	Number of processes that cause I/O congestion (default: 8000).

TABLE I: Rules to determine usage of I/O aggregation.

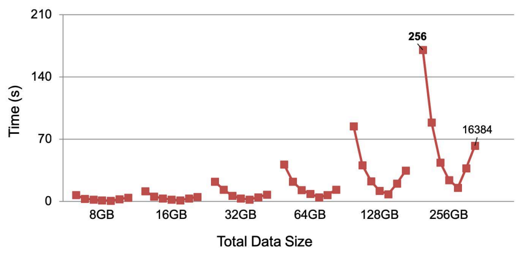
Based on these observations, we propose a rule-based optimization strategy to decide whether to enable or disable I/O aggregation at runtime based on the data access patterns. We list three cases in Table 1 that we found more efficient to disable I/O aggregation and let the clients perform I/O directly.



(a) Contiguous read.



(b) Non-contiguous read from a small number of storage devices.



(c) Non-contiguous read from a large number of storage devices.

Fig. 1: Read time of three cases with different amount of total data (8GB to 256GB). The dotted lines of each data size are the read time by different number of processes, ranging from 256 to 16384.

Case 1 represents a near-contiguous access pattern, where each process reads a large chunk of contiguous data. Due to the high efficiency in reading large chunks of contiguous data, the data transfer overhead between the I/O aggregator and the client will influence the decision making, i.e., when a faster layer of memory or storage, such as shared memory, is unavailable, then having the clients read the data directly without the aggregation would be more efficient. Case 2 represents non-contiguous data accesses and the requested data is stored on a few storage servers. A large number of seeks to non-contiguous locations in storage results in high latency, and makes the communication and data transfer overhead insignificant. Hence, I/O aggregation will not be beneficial and the client processes should directly perform I/O. Case 3 also targets non-contiguous access patterns, but with the requests go to many more storage devices. When there are only a few I/O aggregators and not too many clients, we propose not using I/O aggregation but use more client processes that increases concurrency.

2) Multi-level data caching: Caching data in memory is an effective method to take advantage of the temporal locality of data access. However, an obvious drawback of caching data is the requirement of a large amount of memory space, which may cause out-of-memory errors for applications.

To overcome this shortcoming and utilize the deep memory hierarchy of HPC systems efficiently, we adopt a multi-level data caching mechanism for an ODMS. We use both the compute node's memory and the SSD-based burst buffer layers as cache locations. The ODMS monitors the amount of data cached in the memory layers and automatically transfers them to the next fastest layer (i.e., the burst buffer) when reaching a certain limit. The replacement strategy can use any of the popular policies, such as LRU.

3) Semantics-aware data prefetching: Data prefetching is a widely used technique that exploits the spatial locality of data access. As an ODMS enables storing rich metadata and providing scalable metadata management capabilities, there is an opportunity to perform informed data prefetching using the rich semantic information. Though prefetching adds overhead as more data needs to be read from storage, it can significantly accelerate the overall data access performance when there is a series of data requests.

An ODMS, such as the PDC system, is able to manage the relationships among objects, either automatically recorded by the system, or provided by the user. For example, objects that are from the same container have implied semantic correlation and are likely to be accessed together or sequentially. Other objects that were previously labeled with the same tag are also closely related. With such information available, data can be prefetched proactively based on the current access pattern to reduce future data access latency.

B. Metadata Optimization

Efficient metadata operations also have a major impact on the overall performance, we describe our tuning approaches in the following two categories.

1) Relaxed consistency and aggregation: The task to relax POSIX-IO's strong consistency can be achieved via the metadata management of an ODMS, as all I/O tasks must involve the operating objects' metadata (either retrieve or update) and are performed asynchronously. For example, when an application does not need to read the objects immediately after they are persisted to the storage, as seen in many scientific simulations, it can tolerate a delay in accessing those objects in exchange for better performance.

With relaxed consistency allowed, metadata aggregation becomes an effective approach to reduce the overhead. When multiple clients are sending a large number of metadata requests around the same time, it is desirable to aggregate them into one bulk request and send to the metadata server. Similar to the client-side aggregation, the metadata server can buffer and aggregate the metadata requests, such as requests from the same origin, and process those in bulk for better performance.

2) Asynchronous communication: The metadata operations in existing storage systems are generally blocking (i.e., synchronous), where a client must wait for its requested operation to complete before proceeding to the next task. While performance may be acceptable for existing file-based storage systems, as data access are often within one or a few objects, it is no longer a viable solution for an ODMS. With metadata playing a much more important role in finding and locating data objects, the metadata operations are more frequent and often arrive in bulk. To reduce the metadata access latency and in turn avoid the busy wait, metadata communications can be improved with asynchronous operations. That is, each time a metadata request is sent from the origin, a confirmation is returned back immediately from the target metadata server. Once the metadata server retrieves the requested metadata, it will push the data back to the origin. In this way, multiple metadata requests can be sent and fulfilled concurrently. Note that this is different from the previous PDC asynchronous I/O feature, which applies to the application's data read and write requests and allows it to continue execution while the I/O operation is being performed.

IV. Implementation in the PDC System

We have implemented the proposed optimizations in the PDC system [8], [17], which is an object-centric data management system we have recently developed. In PDC, we offer object-centric APIs that allow asynchronous data movement between memory and storage hierarchy, and provide extensible metadata management. PDC manages data and metadata as objects across the memory hierarchy automatically and transparently. This does not require the expertise and effort from the application developers. By using the client-server approach, the PDC servers actively (asynchronously) move data to its destination, without blocking the client (application) to wait for the I/O to be completed.

PDC currently stores its persistent data in POSIX-based parallel file systems, such as Lustre and DataWarp. Although PDC currently uses these file systems as the backend storage, they are not visible to users and PDC can easily switch to an object store such as DAOS or OpenStack Swift when they become available. Our proposed optimizations are applicable to any ODMS as well. PDC uses the eventual consistency model by default, and can be configured to use strong consistency similar to POSIX-IO through the explicitly locking of an object [17]. By using eventual consistency, several performance bottlenecks are removed and makes PDC a highly scalable framework.

A. I/O Aggregation

PDC servers are user-space processes started by the user, before launching their application. They provide both metadata and data management services. The data service can aggregate requests from the clients, and perform them in batches rather than individually. An I/O request queue is maintained in each server process, with the requests are implemented as remote procedure calls (RPCs). The client request is enqueued upon arrival and a confirmation is sent back immediately afterward to avoid the client to be busy waiting. The server starts processing the requests in the queue when it has aggregated a number of requests (can be set via a hint by the client or after a certain amount of idle time). For read requests, the server needs to retrieve the metadata of their storage locations through the PDC metadata services. The server then decides whether to use I/O aggregation based on the rule-based model we proposed in Section III-A1. If not, it will send the storage metadata back to the client and inform them to perform the read operations instead.

B. Prefetching and Caching

We have implemented prefetching and caching in the PDC system when the server detects repetitive I/O requests. This feature is automatically enabled in PDC and is based on the object relationships provided by the user or observed by the PDC servers. For example, when a majority of the objects in a PDC container have been accessed within a short time, the rest of the objects in that container are automatically prefetched and cached in the server.

With compile time configuration, PDC is able to utilize fast storage layers to cache data, typically either in the memory, in the SSD-based burst buffer, or both. In order to avoid excessive memory consumption, users can specify the upper limit of the memory that the PDC server processes allocate. By default, the (upper) limit is set to be half the total amount of memory in each compute node, so that the server process can cache the same amount of data left available to the client processes on the same node. The user can estimate their application's memory usage and set the limit accordingly, which may be as low as *zero* bytes. When the size of cached data exceeds the limit, data is cached to the next fastest storage layer such as node-local SSDs or a shared burst buffer using the LRU policy.

When I/O aggregation is not used and the client performs the I/O operation, the PDC client library will automatically create a shared memory segment and copies the data there before sending the information to its node-local PDC server. Upon receiving the shared memory information, the server either keeps the data in memory or cache to the Burst Buffer. The overhead of this operation is minimal when the involved data size is not too large (i.e., less than $1GB$). When the involved data is too large and would exceed the available memory, the current PDC system would instruct the clients to perform the I/O operations directly. We are exploring and experimenting further optimizations for this case, such as splitting the large I/O request into smaller ones so that we don't need to allocate the entire shared memory space at once.

C. Metadata Optimizations

1) Asynchronous communication: Similar to the I/O request queue in the data service, PDC server also maintains a metadata request queue for asynchronous metadata request processing. Each time a metadata request is sent from the origin (can be PDC client or server), a confirmation is returned back immediately after it is inserted. When the server finishes the request, it will push the corresponding result back to the origin.

2) Client-side aggregation: A set of collective APIs is provided by PDC for fast metadata retrieval. Based on the number of participating clients and the number of target servers, PDC clients form groups and aggregate their requests to a selected group leader. This significantly reduces the number of requests to the metadata server, and reduces the access latency.

V. Experimental Setup

HPC System	Cori (NERSC)
Storage	Main memory SSD-based burst buffer Hard disk drive (Lustre)
Workloads	BOSS[1]: query and read NyX[2]: write, strong scaling, BD-CATS-IO[3]: read, weak scaling VPIC-IO[3]: write, weak scaling
Comparison	PDC, PDC w/ optimization, HDF5

TABLE II: System configuration and workloads used for evaluating the optimizations.

A. Platform

To demonstrate the effectiveness of our optimizations on the PDC system, we have performed experiments using several applications and I/O kernels with different configurations, and compared with either HDF5, or the original version of PDC, or both.

We evaluated the performance of the PDC system with the new optimizations using a series of experimental configurations [1] [2] [3] for large-scale scientific use cases that are shown in Table II. We ran the PDC system on the "Haswell" partition of the Cori supercomputer located at the National Energy Research Scientific Computing Center (NERSC). Each node of Cori Haswell partition has 32 cores and 128GB memory. Cori is also equipped with a Lustre storage system consists of 248 Object Storage Targets (OSTs), as well as the SSD-based burst buffer. In our experiments, the PDC servers are collocated with the client processes, with a 1:31 client to server ratio, sharing the computing resources of the same node.

To compare the performance, we have measured the total I/O time observed by the application, which is the elapsed time from the first I/O operation to the completion of the last operation, excluding any computation time in between. We compare the original version of PDC (referred as PDC-vanilla) and optimized PDC (PDC-opt) with HDF5, which is a popular I/O library used by various scientific domains. For the PDC system, the time includes overhead such as communication

[1]https://github.com/valiantljk/h5boss
[2]https://github.com/AMReX-Astro/Nyx
[3]https://sdm.lbl.gov/exahdf5/ascr/software.html

and internal data/metadata management. For HDF5, the time includes the file open and close times, along with the time for moving data from memory to a file system, such as Lustre or burst buffer (managed by Cray DataWarp, labeled as "BB" in the plots). HDF5 applications have been compiled with the latest development branch that has various optimizations including avoiding truncate at file close time and collective metadata writes[4]. As the storage systems of Cori are shared by a large number of users, we ran the experiments at least **10** times and reported numbers representing the best results, which got least interference from other HPC users.

B. Implementation of Science Use Cases

1) BOSS: The Baryon Oscillation Spectroscopic Survey (BOSS [18]) data comes from the Sloan Digital Sky Survey (SDSS) project 2, which maps the spatial distribution of galaxies and quasars in the early universe. Each BOSS data object is associated with rich metadata, and can be uniquely identified from three: plate, mjd, and fiber. Plate is the SDSS plug plate ID used to collect the spectrum, mjd is the modified Julian date of the night of the observation, and fiber is the observation's fiber number, ranging from 1 to 1000. We have obtained the BOSS data stored in the HDF5 format, which has $276,575$ files [19]. To measure the performance using the PDC system, we have converted ≈ 25 million BOSS objects into the PDC system.

Data analysis on the BOSS data typically involves queries based on the objects' metadata (plate, mjd, fiber). The existing query code developed by the BOSS management team accepts a list of metadata inputs and copies the matched objects to a new concatenated file [19], which requires a manifest file containing all available BOSS data (stored in HDF5 format) file paths to be specified as the query scope. The code can also be run in parallel, and the list of metadata constraints are divided among the processes.

2) NyX: NyX [20] is an adaptive mesh, massively-parallel, cosmological simulation code that solves equations of compressible hydrodynamics on an adaptive grid hierarchy coupled with an N-body treatment of the dark matter. Particles are evolved via a particle-mesh method, using Cloud-in-Cell deposition/interpolation scheme. Nyx runs produce two types of outputs: Checkpoints and Plotfiles. Checkpoints contain all necessary data to restart the calculation from that time step. Plotfiles are output at user-specified interval or for a list of required (simulation) times; they contain data for post-processing, visualization, and analytics as required by a science case. Although the NyX code generates a large number of AMR boxes during simulation, when writing to the plot file, all AMR boxes from the same level are serialized into a $1D$ array, and is written out as one big contiguous chunk in a single shared file for best performance. Additionally, auxiliary arrays that store the AMR box coordinates and box offsets within the $1D$ array are also written to the HDF5 file.

We have implemented an alternative plot data write function with our PDC APIs. We map each AMR level to a PDC object, and each AMR box from that level becomes a PDC region, with sizes ranging from $32\times32\times32$ to $64\times64\times64$. Each object and region contains the same metadata as those stored in the HDF5 file, the metadata is stored in PDC server's memory and are only persisted to storage system periodically. When fixing the simulation domain size, NyX can be regarded as a **strong scaling** test.

3) BD-CATS-IO: BD-CATS-IO kernel is an I/O kernel that represents the data read patterns of a parallel clustering program that analyze the particle data produced by applications such as VPIC [21]. In this kernel, each MPI process reads a subset of the data with an evenly distributed workload, when increasing the number of processes, it becomes a **weak scaling** test.

4) VPIC-IO: The VPIC-IO kernel is an I/O kernel that simulates the data write behavior by the VPIC [21] application. The data is written into a single shared HDF5 file, and each process writes 8M (8×2^{20}) particles. Each particle has 8 attributes and each process writes out $256MB$ data, and VPIC-IO is a **weak scaling** test.

Both the original BD-CATS-IO and VPIC-IO kernels use HDF5 for performing I/O and are highly tuned using MPI-IO and Lustre optimizations [1], [22]. To demonstrate the performance of PDC system, we have implemented the two kernels using PDC APIs, with each attribute mapping to a PDC object and is associated with various metadata.

VI. Evaluation

A. BOSS

1) Single query: In this set of experiments, we measure the performance of querying objects from whole plates (all 1000 fibers in each plate) and random fibers for only once (where caching and prefetching are disabled). The "whole plate" query results in more contiguous read patterns while the random fiber query leads to a large number of non-contiguous reads. We measure the query time and data read time for both the HDF5 version and our tuned PDC approach. For PDC, under our proposed current rule-based optimization, the whole plate queries will activate the I/O aggregation and the PDC servers read the data. For random fiber queries, the clients perform the read operations after retrieving the storage locations from the PDC metadata servers.

Figures 2 and 3 compare the performance between HDF5 and PDC for random fiber and whole plate queries, with the number of objects from 2560 to over 1 million. For the "random fiber" case in Figure 2, the original PDC would aggregate the I/O requests and perform them on the servers, before transferring back to the clients. Due to its non-contiguous access pattern, the optimized PDC would inform the clients to perform the I/O operations by themselves directly, with improved I/O performance. For the "whole plate" case, the optimized PDC does the same aggregation as the original version, and its results are omitted. In both cases, the tuned PDC offers significantly better performance than HDF5, with the majority improvement resulting in from the advanced

[4]https://bitbucket.hdfgroup.org/projects/HDFFV/repos/hdf5/browse

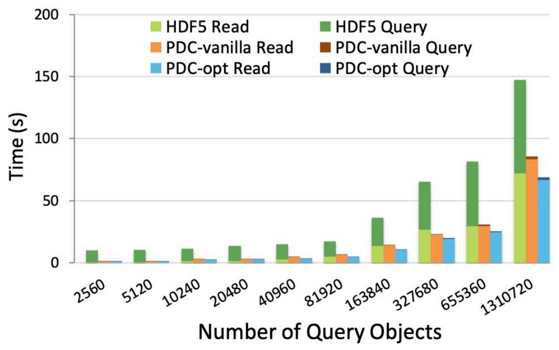


Fig. 2: Time for “random fiber” queries of BOSS dataset with HDF5 and PDC (aggregation disabled, PDC clients read data).

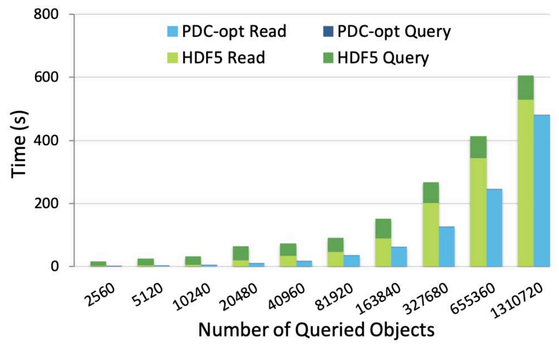


Fig. 3: Time for “whole plate” queries of BOSS dataset with HDF5 and PDC(server aggregation enabled, PDC servers read data).

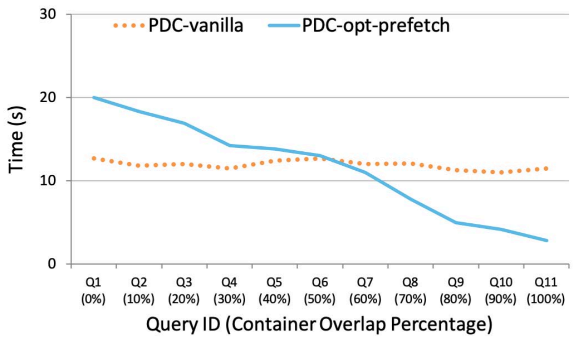


Fig. 4: Time to query and read data with and without prefetching on a sequence of queries, with varying values of container overlap percentages.

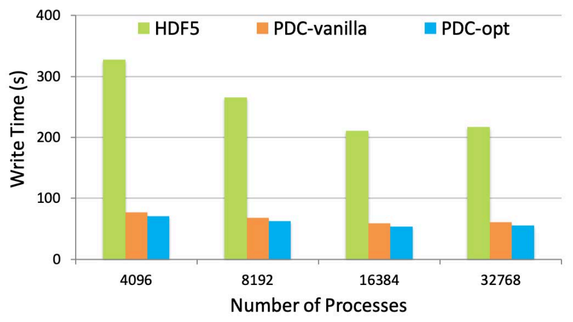


Fig. 5: Total write time observed from the NyX application to write 5 timesteps of AMR data each with $512GB$ to Lustre using HDF5, PDC, and PDC with asynchronous metadata optimization (PDC-opt).

metadata search optimizations. Overall, PDC achieves up to $2\times$ speed up over HDF5.

2) Iterative queries: To further demonstrate the prefetching and caching capabilities, we formulate a sequence of queries and perform them in order and compare with the previous version of PDC without prefetching and caching, as shown in Fig. 4. Since we have already shown that HDF5 is much slower than PDC even without caching in Figures 2 and 3, we do not include the HDF5 results. Each query includes $100k$ objects. For the first query (Q1), we randomly select half objects in a container, so that the prefetch can be activated and the whole container is read and cached by the PDC server. In each of the next queries (Q2 to Q11), we increase the number of objects that are in the same container with previous queries. For example, Q2 has 10% objects that are in the same container as queried objects from Q1, Q3 includes 20% objects from the containers in Q1 and Q2, etc. Due to prefetching requires reading extra data, the first few queries take longer time than the no prefetch version, however, as more data is cached in both memory and the burst buffer, the later queries have significant performance improvement (up to $3\times$) as more data are read from the cache instead of the slower file system.

B. NyX

In Fig. 5, we show the I/O time of both methods that write 5 timesteps of AMR data each with $512GB$ to Lustre. We fixed the total number of application processes for HDF5 and PDC. In PDC's case, we use 1 CPU core per compute node for running I/O services. PDC shows significantly better I/O performance taking advantage of asynchronous I/O feature that hides most of the I/O costs from the application. The compute time is longer than I/O time for all these cases. For PDC, the total time includes creating objects with metadata, sending all 5 asynchronous write requests to PDC servers, and the wait time to write last time step. Overall, PDC achieves up to $4\times$ I/O time speedup compared with the synchronous HDF5 approach. We additionally compare PDC with and without (our previous implementation) asynchronous metadata operation optimization, adding an extra $\approx 11\%$ performance improvement in the PDC optimized case.

C. BD-CATS-IO

To demonstrate PDC's caching capability, we configured the BD-CATS-IO application to read 5 timesteps of particle data with computation time (longer than I/O time) between each timesteps. The read time observed by the application is measured and shown in the figure.

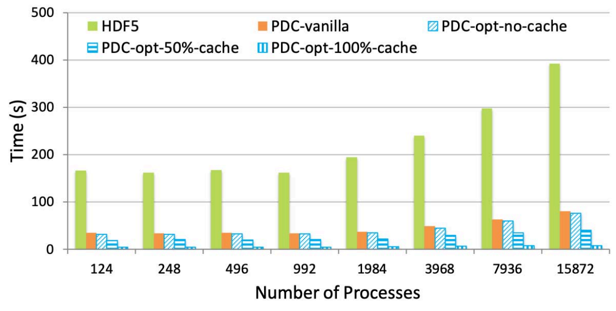


Fig. 6: Read time for BD-CATS-IO with different configurations and number of processes.

Fig. 6 compares the performance for the BD-CATS-IO application to read 5 timesteps of particle data using HDF5 and PDC with different configurations. For PDC, we compared the optimized PDC (PDC-opt*) with as well as the original version (PDC). In addition, we vary the cached data percentage when the application's request can be partially (50%) or fully (100%) fulfilled by PDC. It is a **weak scaling** test with each process reading an equal ($256MB$) amount of data. We vary the number of processes from 4 PDC servers and 124 clients to 64 PDC servers and 1984 clients. In the best case scenario when all requested data are in the cache, PDC can achieve up to $47\times$ speedup over HDF5 and $9\times$ speedup over original PDC.

D. VPIC-IO

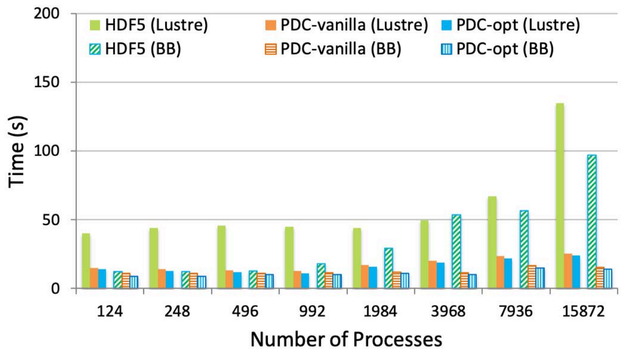


Fig. 7: Time for writing 5 timesteps of VPIC data using HDF5 (synchronous), PDC (asynchronous), and PDC-opt (asynchronous with metadata optimizations).

We compare I/O time of the VPIC-IO kernel using HDF5 and PDC that write to both Lustre and the SSD-based Burst Buffer as shown in Fig. 7. Similar to our previous experiments, PDC achieves $4\times$ speedup because of asynchronous writes. On top of the already highly optimized PDC-vanilla with log-structured write, the optimized PDC achieves another 10% performance improvement with data aggregation and asynchronous metadata optimizations.

VII. Related Work

Object-based storage systems [23] [6] [4][5] [7] have been proposed to overcome the limitations of existing parallel file systems and provide efficient and scalable I/O performance to meet the need of the ever-increasing computing power of high performance computers. Instead of managing data in a hierarchical structure of directories and files, the object storage systems view data as objects in a flat namespace, and each object can be associated with rich metadata.

The advent of object storage systems brings many opportunities and challenges to provide more efficient and scalable I/O performance to the upcoming exa-scale supercomputers. Moving beyond POSIX-IO is among the most significant ones. The POSIX-IO, as part of the POSIX standard [24], defines the file access API, data model, and data consistency semantics. Originally designed for the single node computing systems, it requires strong consistency for data accesses and is often regarded as a major limitation to scalable I/O performance for existing parallel file systems including Lustre [2], PVFS [25], GPFS [3], NFS [26], etc. Several research efforts, such as RADOS [6] and DAOS [7] have focused on relaxing the POSIX semantics and on defining new data models in these systems. However, there are still several optimization aspects, such as semantic-aware prefetching, multi-level caching, I/O aggregation, etc. to be evaluated and integrated into the context of object storage.

Additionally, these object-based storage systems have not sufficiently addressed the challenges in metadata management. Systems such as TagIt [27] and DART [28] do not support asynchronous metadata operations and request aggregation, which could be a performance bottleneck when receiving a large number of requests. Toward an efficient and scalable object data management system, PDC takes advantage of the client-server architecture, the semantic information embedded in objects, and the deep storage hierarchy.

VIII. Conclusions and Future Work

Object-centric data management systems provide new opportunities and challenges toward efficient and scalable data management for the upcoming exa-scale computing. We explored and evaluated various I/O related tuning approaches such as asynchronous data and metadata operations, dynamic I/O workload aggregation, semantic-aware prefetching, multi-level caching, etc. We designed and developed these techniques, and applied to the Proactive Data Containers system. Experimental results demonstrate that these tuning techniques are effective and have a multi-fold performance speedup compared to the original approach.

Our future work includes new methods to intelligently move the data between the memory layers based on dynamic data access pattern analysis, as well as in-transit proactive data analysis that performs the analysis closer to the data storage location.

Acknowledgment

This work is supported by the Director, Office of Science, Office of Advanced Scientific Computing Research, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231 (Project: Proactive Data Containers, Program manager: Dr. Laura Biven). This research used resources of the National Energy Research Scientific Computing Center, which is a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

References

[1] S. Byna, J. Chou, O. Rübel, Prabhat, H. Karimabadi *et al.*, "Parallel I/O, Analysis, and Visualization of a Trillion Particle Simulation," in *Supercomputing*, 2012, pp. 59:1–59:12.

[2] P. J. Braam *et al.*, "The Lustre storage architecture," 2004.

[3] F. B. Schmuck and R. L. Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters." in *FAST*, vol. 2, 2002, pp. 231–244.

[4] Amazon. (2019) Amazon Web Services. Http://s3.amazonaws.com.

[5] J. Arnold, *OpenStack Swift: Using, administering, and developing for swift object storage*. O'Reilly Media, Inc., 2014.

[6] S. A. Weil, A. W. Leung, S. A. Brandt, and C. Maltzahn, "RADOS: A Scalable, Reliable Storage Service for Petabyte-scale Storage Clusters," in *PDSW*, 2007, pp. 35–44.

[7] J. Lofstead, I. Jimenez, C. Maltzahn, Q. Koziol, J. Bent, and E. Barton, "DAOS and Friends: A Proposal for an Exascale Storage System," in *Supercomputing*, 2016, pp. 50:1–50:12.

[8] H. Tang, S. Byna, F. Tessier, T. Wang, B. Dong, J. Mu, Q. Koziol, J. Soumagne, V. Vishwanath, J. Liu *et al.*, "Toward scalable and asynchronous object-centric data management for hpc," in *CCGrid*, 2018, pp. 113–122.

[9] H. Tang, S. Byna, B. Dong, J. Liu, and Q. Koziol, "SoMeta: Scalable Object-centric Metadata Management for High Performance Computing," in *CLUSTER*, 2017, pp. 359–369.

[10] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," *ACM Transactions on Computer Systems (TOCS)*, vol. 10, no. 1, pp. 26–52, 1992.

[11] M. I. Seltzer, K. Bostic, M. K. McKusick, C. Staelin *et al.*, "An implementation of a log-structured file system for unix." in *USENIX Winter*, 1993, pp. 307–326.

[12] J. Bent *et al.*, "Plfs: a checkpoint filesystem for parallel applications," in *Supercomputing*. IEEE, 2009, pp. 1–12.

[13] Q. Liu, J. Logan, Y. Tian *et al.*, "Hello ADIOS: the challenges and lessons of developing leadership class I/O frameworks," *Concurrency and Computation: Practice and Experience*, vol. 26, no. 7, pp. 1453–1473, 2014.

[14] N. Ali, P. Carns, K. Iskra, D. Kimpe, S. Lang, R. Latham, R. Ross, L. Ward, and P. Sadayappan, "Scalable i/o forwarding framework for high-performance computing systems," in *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*. IEEE, 2009, pp. 1–10.

[15] K. Ohta, D. Kimpe, J. Cope, K. Iskra, R. Ross, and Y. Ishikawa, "Optimization techniques at the i/o forwarding layer," in *Cluster Computing (CLUSTER), 2010 IEEE International Conference on*. IEEE, 2010, pp. 312–321.

[16] V. Vishwanath, M. Hereld, K. Iskra, D. Kimpe, V. Morozov, M. E. Papka, R. Ross, and K. Yoshii, "Accelerating i/o forwarding in ibm blue gene/p systems," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society, 2010, pp. 1–10.

[17] J. Mu, J. Soumagne, H. Tang, S. Byna, Q. Koziol, and R. Warren, "A server-managed transparent object storage abstraction for hpc," in *Cluster Computing (CLUSTER), 2018 IEEE International Conference on*, 2018.

[18] K. S. Dawson, D. J. Schlegel, C. P. Ahn, S. F. Anderson, and et al., "The Baryon Oscillation Spectroscopic Survey of SDSS-III," *Astronomical Journal*, vol. 145, p. 10, Jan. 2013.

[19] J. Liu, D. Bard, Q. Koziol, S. Bailey *et al.*, "Searching for millions of objects in the boss spectroscopic survey data with h5boss," in *Scientific Data Summit (NYSDS), 2017 New York*. IEEE, 2017, pp. 1–9.

[20] A. S. Almgren, J. B. Bell, M. J. Lijewski, Z. Lukić, and E. Van Andel, "Nyx: A massively parallel amr code for computational cosmology," *The Astrophysical Journal*, vol. 765, no. 1, p. 39, 2013.

[21] K. J. Bowers, B. J. Albright, L. Yin, B. Bergen, and T. J. T. Kwan, "Ultrahigh Performance Three-dimensional Electromagnetic Relativistic Kinetic Plasma Simulation," *Physics of Plasmas*, vol. 15, no. 5, 2008.

[22] B. Behzad, S. Byna, S. M. Wild, M. Prabhat, and M. Snir, "Improving parallel I/O autotuning with performance modeling," in *HPDC*, 2014, pp. 253–256.

[23] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka, "A cost-effective, high-bandwidth storage architecture," in *ACM SIGOPS operating systems review*, vol. 32, no. 5. ACM, 1998, pp. 92–103.

[24] S. R. Walli, "The POSIX Family of Standards," *StandardView*, vol. 3, no. 1, pp. 11–17, 1995.

[25] P. Carns, W. Ligon III, R. Ross, and R. Thakur, "PVFS: A Parallel Virtual File System for Linux Clusters," *Linux J.*, 2000.

[26] S. Microsystems, "NFS: Network File System Protocol Specification," 1989.

[27] H. Sim, Y. Kim, S. S. Vazhkudai, G. R. Vallée, S.-H. Lim, and A. R. Butt, "Tagit: an integrated indexing and search service for file systems," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2017, p. 5.

[28] W. Zhang, H. Tang, S. Byna, and Y. Cheng, "Dart: Distributed adaptive radix tree for efficient affix-based keyword search on hpc systems," in *The 27th International Conference on Parallel Architectures and Compilation Techniques*, 2018.